Lecture 9: Deadlock

Prof. Ryan Huang
Administrivia

Lab 1 due tonight

If your team decides to use late hours, please fill out the form

Lab 2 released
- Does not depend on lab 1, you can start fresh
- Due October 16th
- Get started!
  - Cannot be done in the last few days
Deadlock

Synchronization is a live gun
- We can easily shoot ourselves in the foot
- Incorrect use of synchronization can block all processes
- You have likely been intuitively avoiding this situation already

If one process tries to access a resource that a second process holds, and vice-versa, they can never make progress

We call this situation **deadlock**, and we’ll look at:
- Definition and conditions necessary for deadlock
- Representation of deadlock conditions
- Approaches to dealing with deadlock
Dining Philosophers Problem

Philosophers spend their lives alternating thinking and eating

Don’t interact with neighbors, occasionally eat
  - Need 2 forks to eat
  - Release both when done

Can only pick up 1 fork at a time
Philosophers in Code (1)

```c
#define N 5 /* number of philosophers */

void philosopher(int i) /* i: philosopher id, 0 to 4 */
{
    while (true) {
        think(); /* philosopher is thinking */
        take_fork(i); /* take left fork */
        take_fork((i + 1) % N); /* take right fork */
        eat(); /* yum-yum, spaghetti */
        put_fork(i); /* put left fork back on the table */
        put_fork((i + 1) % N); /* put right fork back on the table */
    }
}
```
What is a problem with this algorithm?

```c
semaphore forks[N]; /* semaphores for each fork, 
    each initialized to 1 (omitted) */

void take_fork(int i)
{
    forks[i].P(); /* wait for ith fork's semaphore */
}

void put_fork(int i)
{
    forks[i].V(); /* signal ith fork's semaphore */
}
```
How to Avoid Deadlock Here?

Multiple solutions exist

Simple one: allow at most 4 philosophers to sit simultaneously at the table

Another solution: define a partial order for resources (forks)
- Number the forks
- Philosopher must always pick up lower-numbered fork first and then higher-numbered fork
- What happens if four philosophers all pick up their lower-numbered fork?
- Disadvantage
  - Not always practical, when the complete list of all resources is not known in advance

Third solution: all or none each time
2\textsuperscript{nd} Attempt to Dining Philosopher Problem

```c
#define N 5  /* number of philosophers */
#define LEFT (i+N-1) % N  /* i's left neighbor */
#define RIGHT (i+1) % N  /* i's right neighbor */
enum State {THINKING, HUNGRY, EATING};  /* a philosopher's status */
enum State states[N];  /* keep track of each philosopher's status */
semaphore mutex = 1;  /* mutual exclusion for critical section */
semaphore phis[N];  /* semaphore for each philosopher, init to 0 */

void philosopher(int i) /* i: philosopher id, 0 to N-1 */
{
    while (true) {
        think();  /* philosopher is thinking */
        take_forks(i);  /* take both forks */
        eat();  /* yum-yum, spaghetti */
        put_forks(i);  /* put both forks */
    }
}
```
void take_forks(int i) /* i: philosopher id, 0 to N-1 */
{
    mutex.P(); /* enter critical section */
    states[i] = HUNGRY; /* indicate philosopher is hungry */
    test(i); /* try to acquire two forks */
    mutex.V(); /* exit critical section */
    phis[i].P(); /* block if forks not acquired */
}
void put_forks(int i) /* i: philosopher id, 0 to N-1 */
{
    mutex.P(); /* enter critical section */
    states[i] = THINKING; /* indicate i finished eating */
    test(LEFT); /* see if left neighbor can eat now */
    test(RIGHT); /* see if right neighbor can eat now */
    mutex.V(); /* exit critical section */
}

void test(int i) /* i: philosopher id, 0 to N-1 */
{
    if (states[i] == HUNGRY &&
        states[LEFT] != EATING &&
        states[RIGHT] != EATING) {
        states[i] = EATING; /* philosopher i can eat now */
        phis[i].V(); /* signal i to proceed */
    }
}
Notes for the 2nd Attempt Solution

What is the purpose of states array?
- ...given that already have the semaphore array?
- A semaphore doesn’t have operations for checking its value!

What if we don’t use the mutex semaphore?

Why the semaphore array is for each philosopher?
- Our first attempt uses semaphore array for each fork

What if we put phis[i].P(); inside the critical section?

What if we don’t call the two test in put_forks?
Deadlock Definition

Deadlock is a problem that can arise:
- When processes compete for access to limited resources
- When processes are incorrectly synchronized

Definition:
- Deadlock exists among a set of processes if every process is waiting for an event that can be caused only by another process in the set.
mutex_t m1, m2;
void p1(void *ignored) {
    lock(m1);
    lock(m2);
    /* critical section */
    unlock(m2);
    unlock(m1);
}

void p2(void *ignored) {
    lock(m2);
    lock(m1);
    /* critical section */
    unlock(m1);
    unlock(m2);
}
Deadlock Example

Can you have deadlock w/o mutexes?

Same problem with condition variables
- Suppose resource 1 managed by \( c_1 \), resource 2 by \( c_2 \)
- A has 1, waits on \( c_2 \), B has 2, waits on \( c_1 \)

Or w/ combined mutex/condition variable (tricky)

```
lock (a);
lock (b);
while (!ready)
  wait (c, b);
unlock (b);
unlock (a);
```

```
lock (a);
lock (b);
ready = true;
signal (c);
unlock (b);
unlock (a);
```
Deadlock Example

Can you have deadlock w/o mutexes?

Same problem with condition variables

- Suppose resource 1 managed by $c_1$, resource 2 by $c_2$
- A has 1, waits on $c_2$, B has 2, waits on $c_1$

Or with combined mutex/condition variable (tricky)

Lesson: dangerous to hold locks when crossing boundaries!

```
lock (a);
foo(x);
unlock (a);
```

internally uses condition variables

```
lock (a);
bar(y);
unlock (a);
```
Deadlocks w/o Computers

Real issue is resources & how required

E.g., bridge only allows traffic in one direction
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.
Conditions for Deadlock

1. **Mutual exclusion** – At least one resource must be held in a non-sharable mode
2. **Hold and wait** – There must be one process holding one resource and waiting for another resource
3. **No preemption** – Resources cannot be preempted (critical sections cannot be aborted externally)
4. **Circular wait** – There must exist a set of processes \([P_1, P_2, P_3, \ldots, P_n]\) such that \(P_1\) is waiting for \(P_2\), \(P_2\) for \(P_3\), etc.

All of 1–4 necessary for deadlock to occur

Two approaches to dealing with deadlock:

- Pro-active: prevention
- Reactive: detection + corrective action
Prevent by Eliminating One Condition

1. **Mutual exclusion**
   - Buy more resources, split into pieces, or virtualize to make "infinite" copies
   - Threads: threads have copy of registers = no lock

2. **Hold and wait**
   - Wait on all resources at once (must know in advance)

3. **No preemption**
   - Physical memory: virtualized with VM, can take physical page away and give to another process!

4. **Circular wait**
   - Single lock for entire system: (problems?)
   - Partial ordering of resources (next)
Resource Allocation Graph

View system as graph
- Processes and Resources are nodes
- Resource Requests and Assignments are edges

Process:

Resource with 4 instances:

\( P_i \) requesting \( R_j \):

\( P_i \) holding instance of \( R_j \):
Example Resource Allocation Graph
Resource Allocation Graph with Deadlock
Is This Deadlock?
Cycles and Deadlock

If graph has no cycles ⇒ no deadlock

If graph contains a cycle
- Definitely deadlock if only one instance per resource (waits-for graph (WFG))
- Otherwise, maybe deadlock, maybe not

Prevent deadlock with partial order on resources
- e.g., always acquire mutex $m_1$ before $m_2$
- Usually design locking discipline for application this way
Dealing With Deadlock

There are four approaches for dealing with deadlock:

- **Ignore it** – how lucky do you feel?
- **Prevention** – make it impossible for deadlock to happen
- **Avoidance** – control allocation of resources
- **Detection and Recovery** – look for a cycle in dependencies
Deadlock Avoidance

Avoidance

- Provide information in advance about what resources will be needed by processes to guarantee that deadlock will not happen
- System only grants resource requests if it knows that the process can obtain all resources it needs in future requests
- Avoids circularities (wait dependencies)

Tough

- Hard to determine all resources needed in advance
- Good theoretical problem, not as practical to use
Banker’s Algorithm

The Banker’s Algorithm is the classic approach to deadlock avoidance for resources with multiple units.

1. Assign a credit limit to each customer (process).
   - Maximum credit claim must be stated in advance.

2. Reject any request that leads to a dangerous state.
   - A dangerous state is one where a sudden request by any customer for the full credit limit could lead to deadlock.
   - A recursive reduction procedure recognizes dangerous states.

3. In practice, the system must keep resource usage well below capacity to maintain a resource surplus.
   - Rarely used in practice due to low resource utilization.
Detection and Recovery

Detection and recovery

- If we don’t have deadlock prevention or avoidance, then deadlock may occur
- In this case, we need to detect deadlock and recover from it

To do this, we need two algorithms

- One to determine whether a deadlock has occurred
- Another to recover from the deadlock

Possible, but expensive (time consuming)

- Implemented in VMS
- Run detection algorithm when resource request times out
Deadlock Detection

Detection
- Traverse the resource graph looking for cycles
- If a cycle is found, preempt resource (force a process to release)

Expensive
- Many processes and resources to traverse

Only invoke detection algorithm depending on
- How often or likely deadlock is
- How many processes are likely to be affected when it occurs
Deadlock Recovery

Once a deadlock is detected, we have two options…

1. Abort processes
   - Abort all deadlocked processes
     • Processes need to start over again
   - Abort one process at a time until cycle is eliminated
     • System needs to rerun detection after each abort

2. Preempt resources (force their release)
   - Need to select process and resource to preempt
   - Need to rollback process to previous state
   - Need to prevent starvation
Deadlock Summary

Deadlock occurs when processes are waiting on each other and cannot make progress
- Cycles in Resource Allocation Graph (RAG)

Deadlock requires four conditions
- Mutual exclusion, hold and wait, no resource preemption, circular wait

Four approaches to dealing with deadlock:
- Ignore it – Living life on the edge
- Prevention – Make one of the four conditions impossible
- Avoidance – Banker’s Algorithm (control allocation)
- Detection and Recovery – Look for a cycle, preempt or abort
Next time...

Read Chapter 15, 16, 18